# Java内存区域与内存溢出异常

## 2.2运行时数据区域

## 2.3虚拟机对象探秘

对象的创建

当Java虚拟机遇到一条字节码new指令时， 首先将去检查这个指令的参数是否能在常量池中定位到一个类的符号引用， 并且检查这个符号引用代表的类是否已被加载、 解析和初始化过。 如果没有， 那必须先执行相应的类加载过程，

对象的内存布局

在HotSpot虚拟机里， 对象在堆内存中的存储布局可以划分为三个部分： 对象头（Header） 、 实例数据（Instance Data） 和对齐填充（Padding） 。

对齐填充， 这并不是必然存在的， 也没有特别的含义， 它仅仅起着占位符的作用。 由于HotSpot虚拟机的自动内存管理系统要求对象起始地址必须是8字节的整数倍， 换句话说就是任何对象的大小都必须是8字节的整数倍。 对象头部分已经被精心设计成正好是8字节的倍数（1倍或者2倍） ， 因此， 如果对象实例数据部分没有对齐的话， 就需要通过对齐填充来补全。

对象的访问定位

Java程序会通过栈上的reference数据来操作堆上的具体对象。 由于reference类型在《Java虚拟机规范》 里面只规定了它是一个指向对象的引用， 并没有定义这个引用应该通过什么方式去定位、 访问到堆中对象的具体位置， 所以对象访问方式也是由虚拟机实

现而定的， 主流的访问方式主要有使用 句柄和 直接指针两种方式。

## 2.4OutOfMemoryError异常

出现Java堆内存溢出时， 解决这个内存区域的异常， 第一步首先应确认内存中导致OOM的对象是否是必要的， 也就是要先分清楚到底是出现了内存泄漏（Memory Leak） 还是内存溢出（Memory Overflow） 。如果是内存泄漏， 可进一步通过工具查看泄漏对象到GC Roots的引用链， 找到泄漏对象是通过怎样的引用路径、 与哪些GC Roots相关联， 才导致垃圾收集器无法回收它们， 根据泄漏对象的类型信息以及它到GC Roots引用链的信息， 一般可以比较准确地定位到这些对象创建的位置， 进而找出产生内存泄漏的代码的具体位置。

# 垃圾收集器与内存分配策略

# 3.4算法实现细节

**介绍了HotSpot虚拟机如何发起内存回收、 如何加速内存回收， 以及如何保证回收正确性问题。**

### GC Roots枚举

**作用**：

在OopMap的协助下， HotSpot可以快速准确地完成GC Roots枚举。

**详细**：

使用一组称为OopMap【普通对象指针MAP】的数据结构来达到这个目的。在**类加载动作完成的时候**， HotSpot就会把对象内什么偏移量上**是**什么类型的数据计算出来， 在即时编译过程中， 也会在**特定的位置记录**下栈里和寄存器里哪些位置是引用。 这样*收集器在扫描时*就可以直接得知这些信息了， **并不需要真正一个不漏地从方法区等GC Roots开始查找**。

### 安全点

**问题**：

导致OopMap内容变化的指令非常多， 如果为每一条指令都生成对应的OopMap， 那将会需要大

量的额外存储空间， 这样垃圾收集伴随而来的**空间成本就会变得无法忍受的高昂。**

**作用**：

安全点解决如何停顿用户线程， 让虚拟机进入垃圾回收状态的问题。安全点机制保证了程序执行

时， 在不太长的时间内就会遇到可进入垃圾收集过程的安全点。

**详解**：

实际上HotSpot也的确没有为每条指令都生成OopMap， 前面已经提到， 只是在“特定的位置”记录了这些信息， 这些位置被称为**安全点（Safepoint）** 。是强制要求：必须执行到达**安全点后**才能够暂停扫描。

主动式中断的思想是当垃圾收集需要中断线程的时候， 不直接对线程操作， 仅仅简单地设置一个标志位， 各个线程执行过程时会不停地主动去轮询这个标志， 一旦发现中断标志为真时就在自己最近的安全点上主动中断挂起。 **轮询标志的地方和安全点**是重合的， 另外还要加上所有创建对象和其他需要在Java堆上分配内存的地方， 这是为了检查是否即将要发生垃圾收集， 避免没有足够内存分配新对象。

### 安全区域

**问题**：

程序“不执行”的时候呢？ 所谓的程序不执行就是没有分配处理器时间， 典型的场景便是用户线程处于Sleep状态或者Blocked状态， 这时候线程无法响应虚拟机的中断请求， 不能再走到安全的地方去中断挂起自己， 虚拟机也显然不可能持续等待线程重新被激活分配处理器时间。 对于这种情况， 就必须引入安全区域（Safe Region） 来解决？

**作用**：

用户线程处于Sleep状态或者Blocked状态时，通过标识自己进入安全区域,在该区域中任意地方开始垃圾收集都是安全的。

**详解**：

安全区域是指能够确保在某一段代码片段之中， 引用关系不会发生变化， 因此， 在这个区域中任意地方开始垃圾收集都是安全的。 我们也可以把安全区域看作被扩展拉伸了的安全点。

当用户线程执行到安全区域里面的代码时， 首先会标识自己已经进入了安全区域， 那样当这段时间里虚拟机要发起垃圾收集时就不必去管这些已声明自己在安全区域内的线程了。 当线程要离开安全区域时， 它要检查虚拟机是否已经完成了根节点枚举（或者垃圾收集过程中其他需要暂停用户线程的阶段） ， 如果完成了， 那线程就当作没事发生过， 继续执行； 否则它就必须一直等待， 直到收到可以离开安全区域的信号为止。

### 记忆集（Remembered Set）

**问题**：

分代收集理论中，为了解决对象跨代引用所带来的问题， 垃圾收集器在中建立了名为记忆集（Remembered Set） 的数据结构， 用以避免把整个老年代加进GC Roots扫描范围。

**作用**：

使用记忆集（Remembered Set）来缩减GC Roots扫描范围的问题.

**详解：**

记忆集是一种用于记录从非收集区域指向收集区域的指针集合的**抽象数据结构**。在垃圾收集的场景中， 收集器只需要通过**记忆集**判断出某一块非收集区域是否存在有指向了收集区域的指针就可以了， 并不需要了解这些跨代指针的全部细节.

字节数组**CARD\_TABLE**【卡表】的每一个元素都 对应着其标识的 内存区域中一块特定大小的内存块， 这个内存块被称作**“卡页”（Card Page）**

一个卡页的内存中通常包含不止一个对象，只要卡页内有一个（或更多） 对象的字段存在着跨代指针， 那就将对应卡表的数组元素的值标识为1，称为这个元素变脏（Dirty） ， 没有则标识为0。 在垃圾收集发生时， 只要筛选出卡表中变脏的元素， 就能轻易得出哪些卡页内存块中包含跨代指针， 把它们加入GC Roots中一并扫描。

### 写屏障

**问题**：

卡表元素如何维护的？ 例如它们何时变脏。

如何在对象赋值的那一刻去更新维护卡表呢？ 假如是解释执行的字节码， 那相对好处理， 虚拟机负责每条字节码指令的执行， 有充分的介入空间；

但在编译执行的场景中呢？ 经过即时编译后的代码已经是**纯粹的机器指令**流了， 这就必须找到一个**在机器码层面的手段**， 把维护卡表的动作放到每一个赋值操作之中。

**作用**：

通过写屏障（Write Barrier） 技术维护卡表状态的

**详解**：

何时变脏？有其他分代区域中对象引用了本区域对象时， 其对应的卡表元素就应该变脏， 变脏时间点原则上应该发生在引用类型字段赋值的那一刻。

在HotSpot虚拟机里是通过写屏障（Write Barrier） 技术维护卡表状态的，写屏障可以看作在虚拟机层面对“引用类型字段赋值”这个动作的AOP切面。

在引用对象赋值时会产生一个环形（Around） 通知， 供程序执行额外的动作， 也就是说**赋值的前后**都在写屏障的覆盖范畴内。 在赋值前的部分的写屏障叫作写前屏障（Pre-Write Barrier） ， 在赋值后的则叫作写后屏障（Post-Write Barrier） 。

内存屏障（Memory Barrier）是一组处理器指令，用于实现对内存操作的顺序限制.目的是为了指令不因编译优化、 CPU执行优化等原因而导致乱序执行， 它也是可以细分为仅确保读操作顺序正确性和仅确保写操作顺序正确性的内存屏障的。

### 并发的可达性分析

当前主流编程语言的垃圾收集器基本上都是依靠**可达性分析算法**来判定对象是否存活的， 可达性分析算法理论上要求全过程都基于一个**能保障一致性的快照**中才能够进行分析，这意味着**必须全程冻结用户线程的运行**

**为什么必须在一个能保障一致性的快照上才能进行对象图的遍历？**

因为如果用户线程与收集器是并发工作，有可能会把原本存活的对象错误标记为已消亡， 这就是非常致命的后果了， 程序肯定会因此发生错误。

理论上证明了， 当且仅当以下两个条件同时满足时， 会产生“对象消失”的问题， 即原本应该是

黑色的对象被误标为白色：

1.赋值器插入了一条或多条从黑色对象到白色对象的新引用；

2.赋值器删除了全部从灰色对象到该白色对象的直接或间接引用。

**解决方案：**

要解决并发扫描时的对象消失问题， **只需破坏这两个条件的任意一个即可**。 由此分别产生了两种解决方案： 增量更新（Incremental Update） 和原始快照（Snapshot At The Beginning， SATB） 。

**增量更新 要破坏的是第一个条件**， 当黑色对象插入新的指向白色对象的引用关系时， 就将这个新插入的引用记录下来， 等并发扫描结束之后， 再将这些记录过的引用关系中的黑色对象为根， 重新扫描一次。 这可以简化理解为， **黑色对象一旦新插入了指向白色对象的引用之后， 它就变回灰色对象了。**

**原始快照 要破坏的是第二个条件**， 当灰色对象要删除指向白色对象的引用关系时， 就将这个要删除的引用记录下来， 在并发扫描结束之后， 再将这些记录过的引用关系中的灰色对象为根， 重新扫描一次。 这也可以简化理解为， **无论引用关系删除与否， 都会按照刚刚开始扫描那一刻的对象图快照来进行搜索。**

无论是对引用关系记录的插入还是删除， 虚拟机的记录操作**都是通过写屏障实现的**。 在HotSpot虚拟机中， 增量更新和原始快照这两种解决方案都有实际应用， 譬如， CMS是基于增量更新来做并发标记的， G1、 Shenandoah则是用原始快照来实现。

# Class文件结构

